
CS5542 - Lab Assignment 2 – Lab Report

Sub team # 1-2 | Avni Mehta (Class Id: 11) | Raji Muppala (Class Id: 14)

Task 1 Objective:

Build an application to summarize a video by using Clarifai API. Use OpenImg Library to detect key frame images from the Clarify API

Input:

The code is run for three different input videos (<15 second) of McDonald's commercial, cars and kids
Link: https://www.youtube.com/watch?v=0_2I1NVn8vU
https://www.youtube.com/watch?v=s_fuFC7ciI0
https://www.youtube.com/watch?v=F5x-rrjsP0I

Code:

Explanation:

- In KeyFrameDetection.java, import required libraries for Spark, OpenImj and Clarifai API
- In public class KeyFrameDetection, for the given .mkv video, get all the frames from the video.
- For frames extraction, iterate over video frames and select the main frames in the video.
- Compare SIFT features with neighbouring images. When common features < certain threshold, shot the transition.
- Find all the main key points, collect them and output the results as mainframes.
- Connect to the Clarifai API server by using API key and access code for the token.
- Using this connection, access the mainframes file and scan the image in detail and predict information present in the image.
- Update the image with all the possible contents in each image. Output the image.

```java
 */
public class KeyFrameDetection {
    static Video<MBFImage> video;
    //    VideoDisplay<MBFImage> display = VideoDisplay.createVideoDisplay(video);
    static List<MBFImage> imageList = new ArrayList<MBFImage>();
    static List<Long> timeStamp = new ArrayList<Long>();
    static List<Double> mainPoints = new ArrayList<Double>();
    public static void main(String args[]){
        //String path = "input/sample.mkv";
        String path = "input/car.mkv";
        Frames(path);
        MainFrames();
    }

    public static void Frames(String path){
        video = new XuggleVideo(new File(path));
//        VideoDisplay<MBFImage> display = VideoDisplay.createVideoDisplay(video);
        int j=0;
        for (MBFImage mbfImage : video) {
            BufferedImage bufferedFrame = ImageUtilities.createBufferedImageForDisplay(mbfImage
            j++;
            String name = "output/frames/new" + j + ".jpg";
            File outputFile = new File(name);

            try {
                System.out.println("Adding image " + outputFile.getName() );
                ImageIO.write(bufferedFrame, formatName: "jpg", outputFile);

            } catch (IOException e) {
                e.printStackTrace();
            }
```

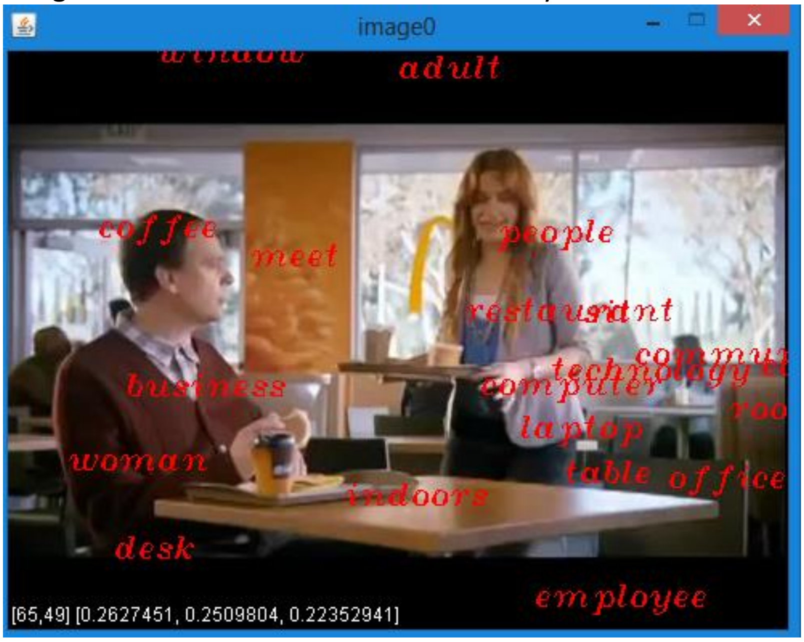
```java
*/
public class ImageAnnotation {
    public static void main(String[] args) throws IOException {
        final ClarifaiClient client = new ClarifaiBuilder( appID: "KKQIegBW9uOl_3vaMSzqq4QCfPNyNBvB7XNBz1vE",  appSecret: "xsY48eiDhhsFo5M7HE3F71ZYkB_tEQmemlWekTgG")
                .client(new OkHttpClient()) // OPTIONAL. Allows customization of OkHttp by the user
                .buildSync(); // or use .build() to get a Future<ClarifaiClient>
        client.getToken();

        File file = new File( pathname: "output/mainframes");
        File[] files = file.listFiles();
        for (int i=0; i<files.length;i++){
            ClarifaiResponse response = client.getDefaultModels().generalModel().predict()
                    .withInputs(
                            ClarifaiInput.forImage(ClarifaiImage.of(files[i]))
                    )
                    .executeSync();
            List<ClarifaiOutput<Concept>> predictions = (List<ClarifaiOutput<Concept>>) response.get();
            MBFImage image = ImageUtilities.readMBF(files[i]);
            int x = image.getWidth();
            int y = image.getHeight();

            System.out.println("*************" + files[i] + "***********");
            List<Concept> data = predictions.get(0).data();
            for (int j = 0; j < data.size(); j++) {
                System.out.println(data.get(j).name() + " - " + data.get(j).value());
                image.drawText(data.get(j).name(), (int)Math.floor(Math.random()*x), (int) Math.floor(Math.random()*y), HersheyFont.ASTROLOGY,  sz: 20, RGBColour.RED);
            }
            DisplayUtilities.displayName(image,  name: "image" + i);
        }

    }
}
```

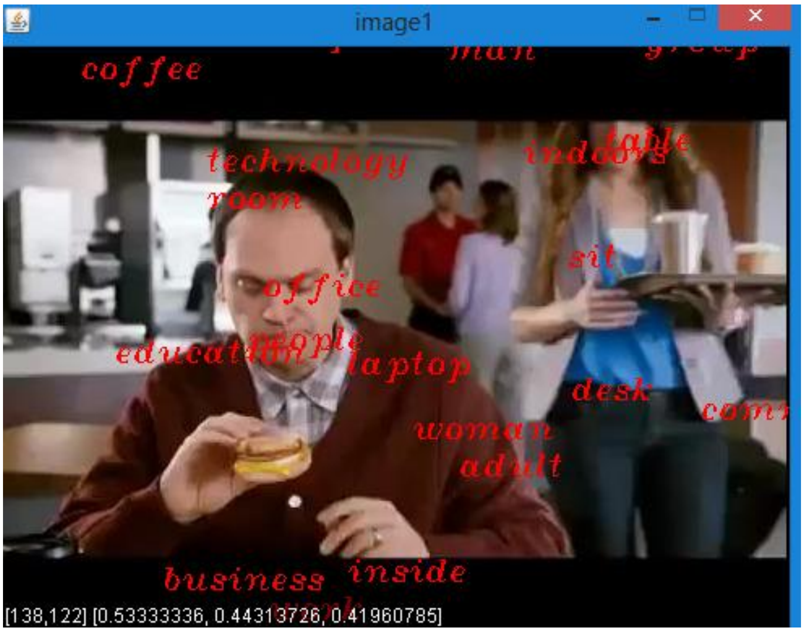
```java
    public static void MainFrames(){
        for (int i=0; i<imageList.size() - 1; i++)
        {
            MBFImage image1 = imageList.get(i);
            MBFImage image2 = imageList.get(i+1);
            DoGSIFTEngine engine = new DoGSIFTEngine();
            LocalFeatureList<Keypoint> queryKeypoints = engine.findFeatures(image1.flatten());
            LocalFeatureList<Keypoint> targetKeypoints = engine.findFeatures(image2.flatten());
            RobustAffineTransformEstimator modelFitter = new RobustAffineTransformEstimator( threshold: 5.0,  nIterations: 1500,
                    new RANSAC.PercentageInliersStoppingCondition( percentageLimit: 0.5));
            LocalFeatureMatcher<Keypoint> matcher = new ConsistentLocalFeatureMatcher2d<Keypoint>(
                    new FastBasicKeypointMatcher<Keypoint>( threshold: 8), modelFitter);
            matcher.setModelFeatures(queryKeypoints);
            matcher.findMatches(targetKeypoints);
            double size = matcher.getMatches().size();
            mainPoints.add(size);
            System.out.println(size);
        }
        Double max = Collections.max(mainPoints);
        for(int i=0; i<mainPoints.size(); i++){
            if(((mainPoints.get(i))/max < 0.01) || i==0){
                Double name1 = mainPoints.get(i)/max;
                BufferedImage bufferedFrame = ImageUtilities.createBufferedImageForDisplay(imageList.get(i+1));
                String name = "output/mainframes/" + i + "_" + name1.toString() + ".jpg";
                File outputFile = new File(name);
                try {
                    ImageIO.write(bufferedFrame,  formatName: "jpg", outputFile);
                } catch (IOException e) {
                    e.printStackTrace();
                }
            }
        }
    }
```

KeyFrameDetection

Output:

Image Annotation for McDonald video Key Frame





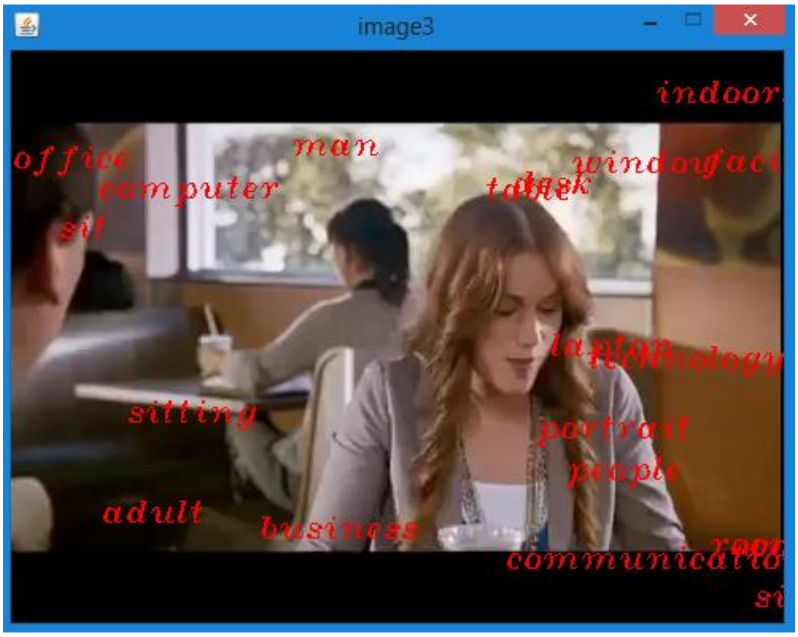


Image Annotation for car video Key Frame

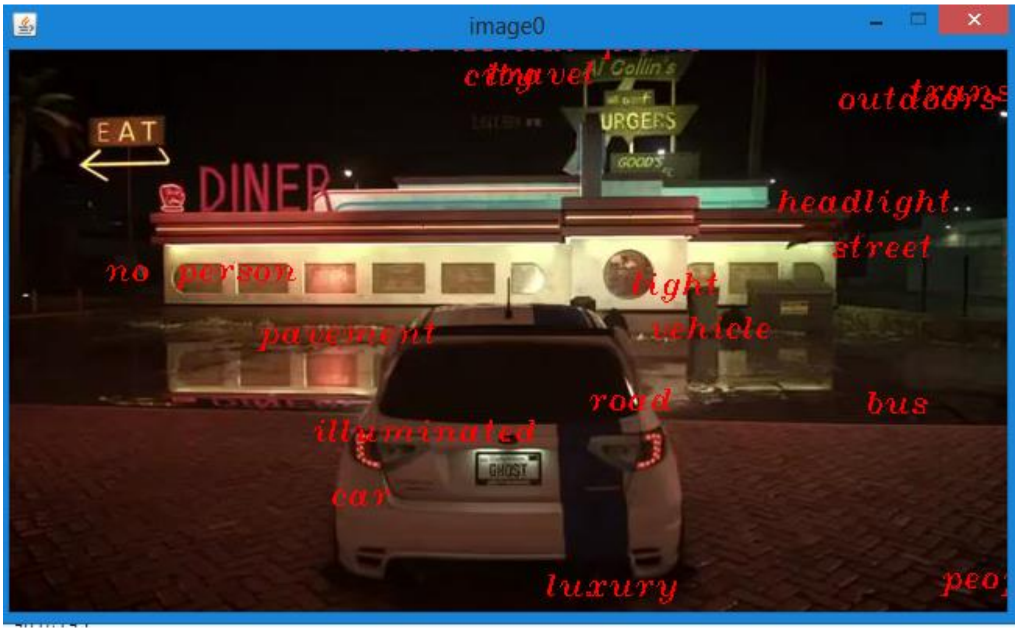
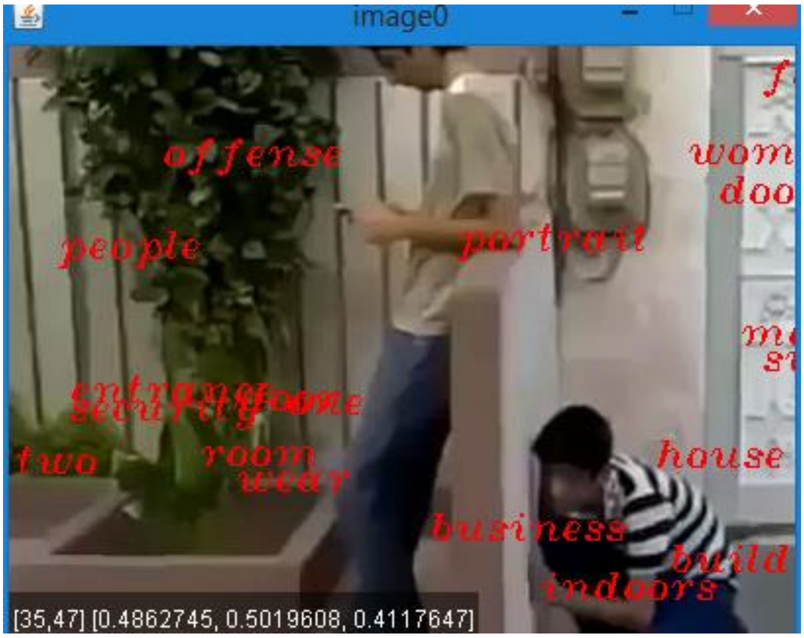


Image Annotation for kids playing video Key Frame



Output on the console

```
ImageAnnotation
"C:\Program Files\Java\jdk1.8.0_161\bin\java" ...
************output\mainframes\241_0.0034100596760443308.jpg**********
adult - 0.996122
woman - 0.9908786
office - 0.99072504
table - 0.9893144
coffee - 0.98511875
computer - 0.9826275
people - 0.9816819
communication - 0.98019475
desk - 0.97984564
laptop - 0.97361076
indoors - 0.9620629
technology - 0.9519186
restaurant - 0.9511918
window - 0.9414147
room - 0.9388094
employee - 0.9377489
meet - 0.9303043
sit - 0.92723155
business - 0.91750723
education - 0.9087678
```

Task 2 Objective:

Classify images related to project and using Random Forest, Decision Tree and Naïve Bayes models and compare their accuracy.

Input:

Images are from MS COCO dataset. We have Identified 4 categories – animals, people, food, interiors. Divided data into train and test4 (for test data).

Code:

a. Random Forest Model:

We have modified the already given code for Random Forest according to the categories based for our VQA project.

DcTreeLab4.scala | IPAppLab4.scala | NaiveBayesLab4.scala | IPSettings.scala

```
import ...

object IPAppLab4 {

  val featureVectorsCluster = new mutable.MutableList[String]

  val IMAGE_CATEGORIES = List("animals","food","Interiors","people")

  /**...*/
  def extractDescriptors(sc: SparkContext, images: RDD[(String, String)]): Unit = {

    if (Files.exists(Paths.get(IPSettings.FEATURES_PATH))) {
      println(s"${IPSettings.FEATURES_PATH} exists, skipping feature extraction..")
      return
    }

    val data = images.map {
      case (name, contents) => {
        val desc = ImageUtils.descriptors(name.split("file:/")(1))
        val list = ImageUtils.matToString(desc)
        println("-- " + list.size)
        list
      }
    }.reduce((x, y) => x ::: y)

    val featuresSeq = sc.parallelize(data)

    featuresSeq.saveAsTextFile(IPSettings.FEATURES_PATH)
    println("Total size : " + data.size)
  }

  def kMeansCluster(sc: SparkContext): Unit = {
```

After importing the required libraries, a cluster vector is created and the image categories is stored as a List. Then we start extracting information from the dataset and save the features from the images.

DcTreeLab4.scala × IPAppLab4.scala × NaiveBayesLab4.scala × IPSettings.scala ×

```scala
def kMeansCluster(sc: SparkContext): Unit = {
  if (Files.exists(Paths.get(IPSettings.KMEANS_PATH))) {
    println(s"${IPSettings.KMEANS_PATH} exists, skipping clusters formation..")
    return
  }

  // Load and parse the data
  val data = sc.textFile(IPSettings.FEATURES_PATH)
  val parsedData = data.map(s => Vectors.dense(s.split(' ').map(_.toDouble)))

  // Cluster the data into {#4} classes using KMeans
  val numClusters = 4
  val numIterations = 20
  val clusters = KMeans.train(parsedData, numClusters, numIterations)

  // Evaluate clustering by computing Within Set Sum of Squared Errors
  val WSSSE = clusters.computeCost(parsedData)
  println("Within Set Sum of Squared Errors = " + WSSSE)

  clusters.save(sc, IPSettings.KMEANS_PATH)
  println(s"Saves Clusters to ${IPSettings.KMEANS_PATH}")
  sc.parallelize(clusters.clusterCenters.map(v => v.toArray.mkString(" "))).saveAsTextFile(IPSettings.KME
}

def createHistogram(sc: SparkContext, images: RDD[(String, String)]): Unit = {
  if (Files.exists(Paths.get(IPSettings.HISTOGRAM_PATH))) {
    println(s"${IPSettings.HISTOGRAM_PATH} exists, skipping histograms creation..")
    return
  }

  val sameModel = KMeansModel.load(sc, IPSettings.KMEANS_PATH)
```

Clusters are processed using Kmeans. The feature data is loaded and parsed. Here, we are clustering the data into 4 classes. ‘Within Set Sum of Squared Errors’ is calculated. Then histogram for all images and reduced features is created.

DcTreeLab4.scala × IPAppLab4.scala × NaiveBayesLab4.scala × IPSettings.scala ×

```scala
def generateRandomForestModel(sc: SparkContext): Unit = {
  if (Files.exists(Paths.get(IPSettings.RANDOM_FOREST_PATH))) {
    println(s"${IPSettings.RANDOM_FOREST_PATH} exists, skipping Random Forest model formation..")
    return
  }

  val data = sc.textFile(IPSettings.HISTOGRAM_PATH)
  val parsedData = data.map { line =>
    val parts = line.split(',')
    LabeledPoint(parts(0).toDouble, Vectors.dense(parts(1).split(' ').map(_.toDouble)))
  }

  // Split data into training (70%) and test (30%).
  val splits = parsedData.randomSplit(Array(0.7, 0.3), seed = 11L)
  val training = splits(0) //parseData
  val test = splits(1)

  // Train a RandomForest model.
  //  Empty categoricalFeaturesInfo indicates all features are continuous.
  val numClasses = 4
  val categoricalFeaturesInfo = Map[Int, Int]()
  val maxBins = 100

  val numOfTrees = 4 to(10, 1)
  val strategies = List("all", "sqrt", "log2", "onethird")
  val maxDepths = 3 to(6, 1)
  val impurities = List("gini", "entropy")

  var bestModel: Option[RandomForestModel] = None
  var bestErr = 1.0
  val bestParams = new mutable.HashMap[Any, Any]()
```

Build the random forest model from the histogram. Data is split into 70% (training) and 30% (testing). Random forest is trained for 4-10 trees. Best error and parameters is printed. The model is saved.

DcTreeLab4.scala | IPAppLab4.scala | NaiveBayesLab4.scala | IPSettings.scala

```scala
  def testImageClassification(sc: SparkContext) = {

    val model = KMeansModel.load(sc, IPSettings.KMEANS_PATH)
    val vocabulary = ImageUtils.vectorsToMat(model.clusterCenters)
    val path = "files/101_ObjectCategories/ant/image_0012.jpg"
    val desc = ImageUtils.bowDescriptors(path, vocabulary)

    val testImageMat = imread(path)
    imshow("Test Image", testImageMat)

    val histogram = ImageUtils.matToVector(desc)

    println("-- Histogram size : " + histogram.size)
    println(histogram.toArray.mkString(" "))

    val nbModel = RandomForestModel.load(sc, IPSettings.RANDOM_FOREST_PATH)
    val p = nbModel.predict(histogram)
    println(s"Predicting test image : " + IMAGE_CATEGORIES(p.toInt))

    waitKey(0)
  }

  /**...*/
  def classifyImage(sc: SparkContext, path: String): Double = {

    val model = KMeansModel.load(sc, IPSettings.KMEANS_PATH)
    val vocabulary = ImageUtils.vectorsToMat(model.clusterCenters)
    val desc = ImageUtils.bowDescriptors(path, vocabulary)
    val histogram = ImageUtils.matToVector(desc)

    println("--Histogram size : " + histogram.size)

```

Test classification using the test images and histogram size. Determine the prediction for the image. Next, generate confusion matrix and print the model accuracy.

b. Naïve Bayes Model:

DcTreeLab4.scala | IPAppLab4.scala | NaiveBayesLab4.scala | IPSettings.scala

```scala
object NaiveBayesLab4 {

  def generateNaiveBayesModel(sc: SparkContext): Unit = {
    // Load and parse the data file.

    if (Files.exists(Paths.get(IPSettings.NAIVE_BAYES_PATH))) {
      println(s"${IPSettings.NAIVE_BAYES_PATH} exists, skipping Naive Bayes model formation..")
      return
    }

    val data = sc.textFile(IPSettings.HISTOGRAM_PATH)
    import java.nio.file.{Files, Paths}
    val parsedData = data.map { line =>
      val parts = line.split(',')
      LabeledPoint(parts(0).toDouble, Vectors.dense(parts(1).split(' ').map(_.toDouble)))
    }

    // Split data into training (70%) and test (30%).
    val splits = parsedData.randomSplit(Array(0.7, 0.3), seed = 11L)
    print("splits size  = " + splits.size)
    val trainingData = splits(0)
    val testData = splits(1)

    val model = NaiveBayes.train(trainingData, lambda = 1.0, modelType = "multinomial")

    // Evaluate model on test instances and compute test error
    val labelAndPreds = testData.map { point =>
      val prediction = model.predict(point.features)
      (point.label, prediction)
    }
    val testErr = labelAndPreds.filter(r => r._1 != r._2).count().toDouble / testData.count()
    println("Test Error = " + testErr)
```

Build the Naïve Bayes model from the histogram. Data is split into 70% for training and 30% for testing. Compute test error by evaluating the model on test instances.

DcTreeLab4.scala × IPAppLab4.scala × NaiveBayesLab4.scala × IPSettings.scala ×

```scala

    System.setProperty("hadoop.home.dir","C:\\Users\\mraje_000\\Documents\\IntelljWorkSpace\\hadoopforspark

    val conf = new SparkConf()
      .setAppName(s"IPApp")
      .setMaster("local[*]")
      .set("spark.executor.memory", "6g")
      .set("spark.driver.memory", "6g")
    val sparkConf = new SparkConf().setAppName("SparkWordCount").setMaster("local[*]")

    val sc=new SparkContext(sparkConf)

    /**
      * From the labeled Histograms a Naive Bayes  Model is created
      */
    generateNaiveBayesModel(sc)

    val testImages = sc.wholeTextFiles(s"${IPSettings.TEST_INPUT_DIR}/*/*.jpg")
    val testImagesArray = testImages.collect()
    var predictionLabels = List[String]()
    testImagesArray.foreach(f => {
      println(f._1)
      val splitStr = f._1.split("file:/")
      val predictedClass: Double = classifyNBImage(sc, splitStr(1))
      val segments = f._1.split("/")
      val cat = segments(segments.length - 2)
      val GivenClass = IMAGE_CATEGORIES.indexOf(cat)
      println(s"Predicting test image : " + cat + " as " + IMAGE_CATEGORIES(predictedClass.toInt))
      predictionLabels = predictedClass + ";" + GivenClass :: predictionLabels
    })

```

Set spark configuration and settings. Run the function that trains the naïve bayes model. Test the model.

DcTreeLab4.scala × IPAppLab4.scala × NaiveBayesLab4.scala × IPSettings.scala ×

```scala
    predictionLabels.foreach(f => {
      val ff = f.split(";")
      println(ff(0), ff(1))
    })
    val predictionLabelsRDD = sc.parallelize(pLArray)


    val pRDD = predictionLabelsRDD.map(f => {
      val ff = f.split(";")
      (ff(0).toDouble, ff(1).toDouble)
    })
    val accuracy = 1.0 * pRDD.filter(x => x._1 == x._2).count() / testImages.count

    println(accuracy)
    ModelEvaluation.evaluateModel(pRDD)
  }

  def classifyNBImage(sc: SparkContext, path: String): Double = {

    val model = KMeansModel.load(sc, IPSettings.KMEANS_PATH)
    val vocabulary = ImageUtils.vectorsToMat(model.clusterCenters)
    val desc = ImageUtils.bowDescriptors(path, vocabulary)
    val histogram = ImageUtils.matToVector(desc)

    println("--Histogram size : " + histogram.size)

    val nbModel = NaiveBayesModel.load(sc, IPSettings.NAIVE_BAYES_PATH)
    val p = nbModel.predict(histogram)

    p
  }
}
```

Test image classification by using test images. Use histogram size to predict the model. Generate confusion matrix and print accuracy.

c. Decision Tree Model:

DcTreeLab4.scala × | IPAppLab4.scala × | NaiveBayesLab4.scala × | IPSettings.scala ×

```scala

object DcTreeLab4 {

  def generateDecissionTreeModel(sc: SparkContext): Unit = {
    // Load and parse the data file.

    if (Files.exists(Paths.get(IPSettings.DECISION_TREE_PATH))) {
      println(s"${IPSettings.DECISION_TREE_PATH} exists, skipping Decession tree model formation..")
      return
    }

    val data = sc.textFile(IPSettings.HISTOGRAM_PATH)
    import java.nio.file.{Files, Paths}
    val parsedData = data.map { line =>
      val parts = line.split(',')
      LabeledPoint(parts(0).toDouble, Vectors.dense(parts(1).split(' ').map(_.toDouble)))
    }

    // Split data into training (70%) and test (30%).
    val splits = parsedData.randomSplit(Array(0.7, 0.3), seed = 11L)
    print("splits size  = " + splits.size)
    val trainingData = splits(0)
    val testData = splits(1)

    // Train a DecisionTree model.
    //  Empty categoricalFeaturesInfo indicates all features are continuous.
    val numClasses = 4
    val categoricalFeaturesInfo = Map[Int, Int]()
    val impurity = "gini"
    val maxDepth = 5
    val maxBins = 32

```

Similar to the Naïve Bayes model, build Decision Tree model from the histogram. Data is split into 70% for training and 30% for testing.

DcTreeLab4.scala × | IPAppLab4.scala × | NaiveBayesLab4.scala × | IPSettings.scala ×

```scala
    val maxBins = 32

    val model = DecisionTree.trainClassifier(trainingData, numClasses, categoricalFeaturesInfo,
      impurity, maxDepth, maxBins)

    // Evaluate model on test instances and compute test error
    val labelAndPreds = testData.map { point =>
      val prediction = model.predict(point.features)
      (point.label, prediction)
    }
    val testErr = labelAndPreds.filter(r => r._1 != r._2).count().toDouble / testData.count()
    println("Test Error = " + testErr)
    println("Learned classification tree model:\n" + model.toDebugString)

    // Save and load model
    model.save(sc, IPSettings.DECISION_TREE_PATH)
    val sameModel = DecisionTreeModel.load(sc, IPSettings.DECISION_TREE_PATH)
  }


  def main(args: Array[String]) {

    System.setProperty("hadoop.home.dir","C:\\Users\\mraje_000\\Documents\\IntelljWorkSpace\\hadoopforspark");

    val conf = new SparkConf()
      .setAppName(s"IPApp")
      .setMaster("local[*]")
      .set("spark.executor.memory", "6g")
      .set("spark.driver.memory", "6g")
    val sparkConf = new SparkConf().setAppName("SparkWordCount").setMaster("local[*]")

    val sc=new SparkContext(sparkConf)
```

Compute test error by evaluating the model on test instances. Set spark configuration and settings.

DcTreeLab4.scala | IPAppLab4.scala | NaiveBayesLab4.scala | IPSettings.scala

```scala
    generateDecissionTreeModel(sc)

    //    testImageClassification(sc)

    val testImages = sc.wholeTextFiles(s"${IPSettings.TEST_INPUT_DIR}/*/*.jpg")
    val testImagesArray = testImages.collect()
    var predictionLabels = List[String]()
    testImagesArray.foreach(f => {
      println(f._1)
      val splitStr = f._1.split("file:/")
      val predictedClass: Double = classifyDCImage(sc, splitStr(1))
      val segments = f._1.split("/")
      val cat = segments(segments.length - 2)
      val GivenClass = IMAGE_CATEGORIES.indexOf(cat)
      println(s"Predicting test image : " + cat + " as " + IMAGE_CATEGORIES(predictedClass.toInt))
      predictionLabels = predictedClass + ";" + GivenClass :: predictionLabels
    })

    val pLArray = predictionLabels.toArray

    predictionLabels.foreach(f => {
      val ff = f.split(";")
      println(ff(0), ff(1))
    })
    val predictionLabelsRDD = sc.parallelize(pLArray)


    val pRDD = predictionLabelsRDD.map(f => {
      val ff = f.split(";")
      (ff(0).toDouble, ff(1).toDouble)
    })
    val accuracy = 1.0 * pRDD.filter(x => x._1 == x._2).count() / testImages.count
```

Run the function that trains the decision tree model. Test the model.

DcTreeLab4.scala | IPAppLab4.scala | NaiveBayesLab4.scala | IPSettings.scala

```scala
    val predictionLabelsRDD = sc.parallelize(pLArray)


    val pRDD = predictionLabelsRDD.map(f => {
      val ff = f.split(";")
      (ff(0).toDouble, ff(1).toDouble)
    })
    val accuracy = 1.0 * pRDD.filter(x => x._1 == x._2).count() / testImages.count

    println(accuracy)
    ModelEvaluation.evaluateModel(pRDD)
  }

  def classifyDCImage(sc: SparkContext, path: String): Double = {

    val model = KMeansModel.load(sc, IPSettings.KMEANS_PATH)
    val vocabulary = ImageUtils.vectorsToMat(model.clusterCenters)
    val desc = ImageUtils.bowDescriptors(path, vocabulary)
    val histogram = ImageUtils.matToVector(desc)

    println("--Histogram size : " + histogram.size)

    val nbModel = DecisionTreeModel.load(sc, IPSettings.DECISION_TREE_PATH)
    val p = nbModel.predict(histogram)

    p
  }
}
```

Test image classification by using test images. Use histogram size to predict the model. Generate confusion matrix and print accuracy

Output:

<u>Output Observation</u>:

Model	Accuracy	Confusion Matrix

Random Forest	69.74%	`|=================== Confusion matrix`
		`27.0 2.0 1.0 6.0`
		`3.0 10.0 0.0 3.0`
		`3.0 0.0 8.0 1.0`
		`1.0 2.0 1.0 8.0`

Decision Tree	55.26%	`|=================== Confusion matrix :`
		`26.0 0.0 9.0 1.0`
		`5.0 6.0 3.0 2.0`
		`6.0 0.0 6.0 0.0`
		`4.0 0.0 4.0 4.0`

Naïve Bayes	47.39%	`|=================== Confusion matrix =`
		`36.0 0.0 0.0 0.0`
		`16.0 0.0 0.0 0.0`
		`12.0 0.0 0.0 0.0`
		`12.0 0.0 0.0 0.0`

Random Forest model has highest accuracy as compared to Decision Tree model and Naïve Bayes Model. As expected, the Naïve Bayes model accuracy is lowest amongst the three. Overall the accuracy is in the lower range because a subset of the dataset is used.